EXHIBIT K

[Top] [Contents] [Index] [?]

GNU Classpath Hacker's Guide

This file contains important information you will need to know if you are going to hack on the GNU Classpath project code.

This document contains important information you'll want to know if you want to hack on GNU Classpath, Essential Libraries for Java, to help create free core class libraries for use with virtual machines and compilers for the java programming language.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

1. Introduction

The GNU Classpath Project is dedicated to providing a 100% free, clean room implementation of the standard core class libraries for compilers and runtime environments for the java programming language. It offers free software developers an alternative core library implementation upon which larger java-like programming environments can be built. The GNU Classpath Project was started in the Spring of 1998 as an official Free Software Foundation project. Most of the volunteers working on GNU Classpath do so in their spare time, but a couple of projects based on GNU Classpath have paid programmers to improve the core libraries. We appreciate everyone's efforts in the past to improve and help the project and look forward to future contributions by old and new members alike.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

2. Requirements

Although GNU Classpath is following an open development model where input from developers is welcome, there are certain base requirements that need to be met by anyone who wants to contribute code to this project. They are mostly dictated by legal requirements and are not arbitrary restrictions chosen by the GNU Classpath team.

You will need to adhere to the following things if you want to donate code to the GNU Classpath project:

- **Never under any circumstances refer to proprietary code while working on GNU Classpath.** It is best if you have never looked at alternative proprietary core library code at all. To reduce temptation, it would be best if you deleted the `src.zip' file from your proprietary JDK distribution (note that recent versions of GNU Classpath and the compilers and environments build on it are mature enough to not need any proprietary implementation at all when working on GNU Classpath, except in exceptional cases where you need to test compatibility issues pointed out by users). If you have signed Sun's non-disclosure statement, then you unfortunately cannot work on

Classpath code at all. If you have any reason to believe that your code might be "tainted", please say something on the mailing list before writing anything. If it turns out that your code was not developed in a clean room environment, we could be very embarrassed someday in court. Please don't let that happen.

- **Never decompile proprietary class library implementations.** While the wording of the license in Sun's Java 2 releases has changed, it is not acceptable, under any circumstances, for a person working on GNU Classpath to decompile Sun's class libraries. Allowing the use of decompilation in the GNU Classpath project would open up a giant can of legal worms, which we wish to avoid.
- Classpath is licensed under the terms of the GNU General Public License, with a special exception included to allow linking with non-GPL licensed works as long as no other license would restrict such linking. To preserve freedom for all users and to maintain uniform licensing of Classpath, we will not accept code into the main distribution that is not licensed under these terms. The exact wording of the license of the current version of GNU Classpath can be found online from the GNU Classpath license page and is of course distributed with current snapshot release from ftp://ftp.gnu.org/gnu/classpath/ or by obtaining a copy of the current CVS tree.
- GNU Classpath is GNU software and this project is being officially sponsored by the Free Software Foundation. Because of this, the FSF will hold copyright to all code developed as part of GNU Classpath. This will allow them to pursue copyright violators in court, something an individual developer may neither have the time nor resources to do. Everyone contributing code to GNU Classpath will need to sign a copyright assignment statement. Additionally, if you are employed as a programmer, your employer may need to sign a copyright waiver disclaiming all interest in the software. This may sound harsh, but unfortunately, it is the only way to ensure that the code you write is legally yours to distribute.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

3. Volunteering to Help

The GNU Classpath project needs volunteers to help us out. People are needed to write unimplemented core packages, to test GNU Classpath on free software programs written in the java programming language, to test it on various platforms, and to port it to platforms that are currently unsupported.

While pretty much all contributions are welcome (but see see section Requirements) it is always preferable that volunteers do the whole job when volunteering for a task. So when you volunteer to write a Java package, please be willing to do the following:

- Implement a complete drop-in replacement for the particular package. That means implementing any "internal" classes. For example, in the java.net package, there are non-public classes for implementing sockets. Without those classes, the public socket interface is useless. But do not feel obligated to completely implement all of the functionality at once. For example, in the java.net package, there are different types of protocol handlers for different types of URLs. Not all of these need to be written at once.
- Please write complete and thorough API documentation comments for every public and protected method and variable. These should be superior to Sun's and cover everything about the item being documented.
- Please write a regression test package that can be used to run tests of your package's functionality. GNU Classpath uses the Mauve project for testing the functionality of the core class libraries. The Classpath Project is fast approaching the point in time where all modifications to the source code repository will require appropriate test cases in Mauve to ensure correctness and prevent

regressions.

Writing good documentation, tests and fixing bugs should be every developer's top priority in order to reach the elusive release of version 1.0.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

4. Project Goals

The goal of the Classpath project is to produce a free implementation of the standard class library for Java. However, there are other more specific goals as to which platforms should be supported.

Classpath is targeted to support the following operating systems:

1. Free operating systems. This includes GNU/Linux, GNU/Hurd, and the free BSDs.
2. Other Unix-like operating systems.
3. Platforms which currently have no Java support at all.
4. Other platforms such as MS-Windows.

While free operating systems are the top priority, the other priorities can shift depending on whether or not there is a volunteer to port Classpath to those platforms and to test releases.

Eventually we hope the Classpath will support all JVMs that provide JNI or CNI support. However, the top priority is free JVMs. A small list of Compiler/VM environments that are currently actively incorporating GNU Classpath is below. A more complete overview of projects based on GNU classpath can be found online at the GNU Classpath stories page.

1. GCJ
2. jamvm
3. cacao
4. Jikes RVM
5. Kaffe
6. IKVM

As with OS platform support, this priority list could change if a volunteer comes forward to port, maintain, and test releases for a particular JVM. Since gcj is part of the GNU Compiler Collective it is one of the most important targets. But since it doesn't currently work out of the box with GNU Classpath it is not the easiest target. When hacking on GNU Classpath the easiest solution is to use compilers and runtime environments that work out of the box with it, such as the Eclipse compiler, ecj, and the runtime environments jamvm and cacao. Both Jikes RVM and Kaffe use an included version of GNU Classpath by default, but Kaffe can now use a pre-installed version and Jikes RVM supports using a CVS snapshot as well as the latest release. Working directly with targets such as Jikes RVM, gcj and IKVM is possible but can be a little more difficult as changes have to be merged back into GNU Classpath proper, which requires additional work. Due to a recent switch to the use of 1.5 language features within GNU Classpath, a compiler compatible with these features is required. At present, this includes the Eclipse compiler, ecj, and the OpenJDK compiler.

GNU Classpath currently implements the majority of the 1.4 and 1.5 APIs (binary compatibility is above

95% for both, but does not take into account internal implementations of features such as graphic and sound support). There is support for some 1.6 APIs but this is still nascent. Please do not create classes that depend on features in other packages unless GNU Classpath already contains those features. GNU Classpath has been free of any proprietary dependencies for a long time now and we like to keep it that way. Finishing, polishing up, documenting, testing and debugging current functionality is of higher priority then adding new functionality.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

5. Needed Tools and Libraries

If you want to hack on Classpath, you should at least download and install the following tools and try to familiarize yourself with them. In most cases having these tools installed will be all you really need to know about them. Also note that when working on (snapshot) releases only a 1.5 compiler (plus a free VM from the list above and the libraries listed below) is required. The other tools are only needed when working directly on the CVS version.

- GNU make 3.80+
- GCC 2.95+
- Eclipse Compiler for Java 3.1+
- CVS 1.11+
- automake 1.9+
- autoconf 2.59+
- libtool 1.5+
- GNU m4 1.4
- texinfo 4.2+

All of these tools are available from gnudist.gnu.org via anonymous ftp, except CVS which is available from www.cvshome.org and the Eclipse Compiler for Java, which is available from www.eclipse.org/jdt/core.

Except for the Eclipse Compiler for Java, they are fully documented with texinfo manuals. Texinfo can be browsed with the Emacs editor, or with the text editor of your choice, or transformed into nicely printable Postscript.

Here is a brief description of the purpose of those tools.

make

GNU make ("gmake") is required for building Classpath.

GCC

The GNU Compiler Collection. This contains a C compiler (gcc) for compiling the native C code and a compiler for the java programming language (gcj). You will need at least gcc version 2.95 or higher in order to compile the native code. There is currently no released version of gcj that can compile the Java 1.5 programming language used by GNU Classpath.

ecj

The Eclipse Compiler for Java. This is a compiler for the Java 1.5 programming language. It translates source code to bytecode. The Eclipse Foundation makes "ecj.jar" available as the JDT Core Batch Compiler download.

CVS

A version control system that maintains a centralized Internet repository of all code in the Classpath system.

automake

This tool automatically creates `Makefile.in' files from `Makefile.am' files. The `Makefile.in' is turned into a `Makefile' by `autoconf'.

Why use this? Because it automatically generates every makefile target you would ever want (`clean', `install', `dist', etc) in full compliance with the GNU coding standards. It also simplifies Makefile creation in a number of ways that cannot be described here. Read the docs for more info.

autoconf

Automatically configures a package for the platform on which it is being built and generates the Makefile for that platform.

libtool

Handles all of the zillions of hairy platform specific options needed to build shared libraries.

m4

The free GNU replacement for the standard Unix macro processor. Proprietary m4 programs are broken and so GNU m4 is required for autoconf to work though knowing a lot about GNU m4 is not required to work with autoconf.

perl

Larry Wall's scripting language. It is used internally by automake.

texinfo

Manuals and documentation (like this guide) are written in texinfo. Texinfo is the official documentation format of the GNU project. Texinfo uses a single source file to produce output in a number of formats, both online and printed (dvi, info, html, xml, etc.). This means that instead of writing different documents for online information and another for a printed manual, you need write only one document. And when the work is revised, you need revise only that one document.

For any build environment involving native libraries, recent versions of `autoconf`, `automake`, and `libtool` are required if changes are made that require rebuilding `configure', `Makefile.in',

`aclocal.m4', or `config.h.in'.

When working from CVS you can run those tools by executing `autogen.sh` in the source directory.

For building the Java bytecode (.class files), you can select which compiler should be employed using `--with-javac' or `--with-ecj' as an argument to `configure`; the present default is `ecj` if found.

Instead of `ecj`, you can also use `javac`, which is available at openjdk.dev.java.net/compiler.

For compiling the native AWT libraries you need to have the following libraries installed (unless `--disable-gtk-peer' is used as an argument to `configure`):

GTK+ 2.8.x

GTK+ is a multi-platform toolkit for creating graphical user interfaces. It is used as the basis of the GNU desktop project GNOME.

gdk-pixbuf

gdk-pixbuf is a GNOME library for representing images.

XTest

www.x.org hosts the XTest Extension (libXtst). It is necessary for GdkRobot support in java.awt.

There is a bug in earlier versions of at-spi, atk, and gail, which are used for GNOME accessibility. Prior to version 1.18.0 of these packages, gtk graphical applications should be run without accessibility (clear the GTK_MODULES environment variable).

For building the Qt AWT peer JNI native libraries you have to specify `--enable-qt-peer' and need the following library:

Qt

Qt version 4.0.1 or higher. The Qt library is a cross-platform graphics toolkit.

Please note that at the moment most operating systems do not ship Qt version 4.0.1 by default. We recommend using GNU Classpath' Qt support only for its developers and bug reporters. See the wiki for details on how to get it to work.

For building the X AWT peers you have to specify where to find the Escher library on your system using the `--with-escher=ABS.PATH' option. You will need the following library:

Escher

Escher version 0.2.3 or higher. The Escher library is an implementation of X protocol and associated libraries written in the Java programming language.

For building the ALSA midi provider code you will need the following library:

ALSA

ALSA libraries.

The ALSA project provides sound device drivers and associated libraries for the Linux kernel.

Building the ALSA midi provider code can be disabled by passing `--disable-alsa' to `configure`.

For building the DSSI midi synthesizer provider code you will need the following libraries:

DSSI

DSSI library for audio processing plugins.

liblo

liblo, the Lightweight OSC implementation.

LADSPA

LADSPA, the Linux Audio Developer's Simple Plugin API.

JACK

JACK, a low latency audio server.

libsndfile

libsndfile, an audio file I/O library.

fluidsynth

fluidsynth, a real-time SoundFont 2 based soft-synth.

The GConf-based backend for java.util.prefs needs the following library headers:

GConf

GConf version 2.6.0 (or higher). GConf is used for storing desktop and application configuration settings in GNOME.

The GStreamer backend for javax.sound.sampled (The Java Sound API, not including the MIDI portion) needs the following library headers:

GStreamer

GStreamer version 0.10.10 (or higher). You will also need at least gstreamer-base and gstreamer-plugins-base. More plugins can be used to allow streaming of different sound types but are not a compile time requirement. See README.gstreamer in the source distribution for more informations.

For building `gcjwebplugin` you'll need the Mozilla plugin support headers and libraries, which are available at www.mozilla.org.

For enabling the com.sun.tools.javac support in tools.zip you will need a jar file containing the Eclipse Java Compiler. Otherwise com.sun.tools.javac will not be included in `tools.zip'.

For building the xmlj JAXP implementation (disabled by default, use `configure --enable-xmlj`) you need the following libraries:

libxml2

libxml2 version 2.6.8 or higher.

The libxml2 library is the XML C library for the Gnome desktop.

libxslt

libxslt version 1.1.11 or higher.

The libxslt library if the XSLT C library for the Gnome desktop.

GNU Classpath comes with a couple of libraries included in the source that are not part of GNU Classpath proper, but that have been included to provide certain needed functionality. All these external libraries should be clearly marked as such. In general we try to use as much as possible the clean upstream versions of these sources. That way merging in new versions will be easier. You should always try to get bug fixes to these files accepted upstream first. Currently we include the following 'external' libraries. Most of these sources are included in the `external' directory. That directory also contains a `README' file explaining how to import newer versions.

JSR166 concurrency support

Can be found in `external/jsr166'. Provides java.util.concurrent and its subpackages. Upstream is Doug Lea's Concurrency Interest Site.

RelaxNG Datatype Interfaces

Can be found in `external/relaxngDatatype'. Provides org.relaxng.datatype and its subpackages. Upstream is http://www.oasis-open.org/committees/relax-ng/.

Simple API for XML (SAX)

Can be found in `external/sax'. Provides org.xml.sax and its subpackages. Upstream is http://www.saxproject.org.

Document Object Model (DOM) bindings

Can be found in `external/w3c_dom'. Provides org.w3c.dom and its subpackages. Upstream locations are listed in `external/w3c_dom/README'.

fdlibm

Can be found in `native/fdlibm'. Provides native implementations of some of the Float and Double operations. Upstream is libgcj, they sync again with the 'real' upstream http://www.netlib.org/fdlibm/readme. See also java.lang.StrictMath.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

6. Installation instructions

This package was designed to use the GNU standard for configuration and makefiles. To build and install do the following:

1. Configuration

   Run the `configure` script to configure the package. There are various options you might want to pass to `configure` to control how the package is built. Consider the following options, `configure --help` gives a complete list.

   `--enable-java'

   compile Java source (default=`yes').

   `--enable-jni'

   compile JNI source (default=`yes').

   `--enable-gtk-peer'

   compile GTK native peers (default=`yes').

   `--enable-qt-peer'

   compile Qt4 native peers (default=`no').

   `--enable-default-toolkit'

   fully qualified class name of default AWT toolkit (default=`no').

   `--enable-xmlj'

   compile native libxml/xslt library (default=`no').

   `--enable-load-library'

   enable to use JNI native methods (default=`yes').

   `--enable-local-sockets'

enable build of local Unix sockets.

`--with-glibj'

define what to install `(zip|flat|both|none)' (default=`zip').

`--with-escher=/path/to/escher'

enable build of the X/Escher peers, with the escher library at `/path/to/escher', either in the form of a JAR file, or a directory containing the .class files of Escher.

`--enable-Werror'

whether to compile C code with `-Werror' which turns any compiler warning into a compilation failure (default=`no').

`--with-gjdoc'

generate documentation using `gjdoc` (default=`no').

`--with-jay'

Regenerate the parsers with `jay`, must be given the path to the `jay` executable

`--with-glibj-zip=ABS.PATH'

use prebuilt glibj.zip class library

`--with-ecj-jar=ABS.PATH'

specify jar file containing the Eclipse Java Compiler

`--with-gstreamer-peer'

build the experimental GStreamer peer (see `README.gstreamer')

For more flags run `configure --help`.

2. Building

Type `gmake` to build the package. There is no longer a dependency problem and we aim to keep it that way.

3. Installation

Type `gmake install` to install everything. This may require being the superuser. The default install path is /usr/local/classpath you may change it by giving `configure` the `--prefix=<path>' option.

Report bugs to classpath@gnu.org or much better to the GNU Classpath bug tracker at Savannah.

Happy Hacking!

Once installed, GNU Classpath is ready to be used by any VM that supports using the official version of GNU Classpath. Simply ensure that `/usr/local/classpath/share/classpath' is in your `CLASSPATH' environment variable. You'll also have to set your `LD_LIBRARY_PATH' variable (or similar system configuration) to include the Classpath native libraries in `/usr/local/classpath/lib/classpath'.

NOTE All example paths assume the default prefix is used with `configure'. If you don't know what this means then the examples are correct.

```
LD_LIBRARY_PATH=/usr/local/classpath/lib/classpath
CLASSPATH=/usr/local/classpath/share/classpath/glibj.zip:.
export LD_LIBRARY_PATH CLASSPATH
```

More information about the VMs that use GNU Classpath can be found in the `README' file.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

7. Building and running with the X AWT peers

In order build the X peers you need the Escher library version 0.2.3 from escher.sourceforge.net. Unpack (and optionally build) the Escher library following the instructions in the downloaded package. Enable the build of the X peers by passing `--with-escher=/path/to/escher' to `configure' where `/path/to/escher' either points to a directory structure or JAR file containing the Escher classes. For Unix systems it is preferable to also build local socket support by passing `--enable-local-sockets', which accelerates the network communication to the X server significantly.

In this release you have to enable the X peers at runtime by setting the system property awt.toolkit=gnu.java.awt.peer.x.XToolkit by passing `-Dawt.toolkit=gnu.java.awt.peer.x.XToolkit' to the `java' command when running an application.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

8. Misc. Notes

Compilation is accomplished using a compiler's @file syntax. For our part, we avoid placing make style dependencies as rules upon the compilation of a particular class file and leave this up to the Java compiler instead.

The `--enable-maintainer-mode' option to `configure' currently does very little and shouldn't be used by ordinary developers or users anyway.

On Windows machines, the native libraries do not currently build, but the Java bytecode library will. GCJ trunk is beginning to work under Cygwin.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

9. Programming Standards

For C source code, follow the GNU Coding Standards. The standards also specify various things like the install directory structure. These should be followed if possible.

For Java source code, please follow the GNU Coding Standards, as much as possible. There are a number of exceptions to the GNU Coding Standards that we make for GNU Classpath as documented in this guide. We will hopefully be providing developers with a code formatting tool that closely matches those rules soon.

For API documentation comments, please follow How to Write Doc Comments for Javadoc. We would like to have a set of guidelines more tailored to GNU Classpath as part of this document.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

9.1 Java source coding style

Here is a list of some specific rules used when hacking on GNU Classpath java source code. We try to follow the standard GNU Coding Standards for that. There are lots of tools that can automatically generate it (although most tools assume C source, not java source code) and it seems as good a standard as any. There are a couple of exceptions and specific rules when hacking on GNU Classpath java source code however. The following lists how code is formatted (and some other code conventions):

- Java source files in GNU Classpath are encoded using UTF-8. However, ordinarily it is considered best practice to use the ASCII subset of UTF-8 and write non-ASCII characters using \u escapes.
- If possible, generate specific imports (expand) over java.io.* type imports. Order by gnu, java, javax, org. There must be one blank line between each group. The imports themselves are ordered alphabetically by package name. Classes and interfaces occur before sub-packages. The classes/interfaces are then also sorted alphabetical. Note that uppercase characters occur before lowercase characters.

```
import gnu.java.awt.EmbeddedWindow;

import java.io.IOException;
import java.io.InputStream;

import javax.swing.JFrame;
```

- Blank line after package statement, last import statement, classes, interfaces, methods.
- Opening/closing brace for class and method is at the same level of indent as the declaration. All other braces are indented and content between braces indented again.
- Since method definitions don't start in column zero anyway (since they are always inside a class definition), the rational for easy grepping for "^method_def" is mostly gone already. Since it is customary for almost everybody who writes java source code to put modifiers, return value and method name on the same line, we do too.
- Implements and extends on separate lines, throws too. Indent extends, implements, throws. Apply

deep indentation for method arguments.

- Don't add a space between a method or constructor call/definition and the open-bracket. This is because often the return value is an object on which you want to apply another method or from which you want to access a field.

  Don't write:

  ```
  getToolkit ().createWindow (this);
  ```

  But write:

  ```
  getToolkit().createWindow(this);
  ```

- The GNU Coding Standard it gives examples for almost every construct (if, switch, do, while, etc.). One missing is the try-catch construct which should be formatted as:

  ```
  try
    {
      //
    }
  catch (...)
    {
      //
    }
  ```

- Wrap lines at 80 characters after assignments and before operators. Wrap always before extends, implements, throws, and labels.
- Don't put multiple class definitions in the same file, except for inner classes. File names (plus .java) and class names should be the same.
- Don't catch a `NullPointerException` as an alternative to simply checking for `null`. It is clearer and usually more efficient to simply write an explicit check.

  For instance, don't write:

  ```
  try
    {
      return foo.doit();
    }
  catch (NullPointerException _)
    {
      return 7;
    }
  ```

  If your intent above is to check whether `` `foo' `` is `null`, instead write:

  ```
  if (foo == null)
    return 7;
  else
    return foo.doit();
  ```

- Don't use redundant modifiers or other redundant constructs. Here is some sample code that shows various redundant items in comments:

  ```
  /*import java.lang.Integer;*/
  /*abstract*/ interface I {
     /*public abstract*/ void m();
     /*public static final*/ int i = 1;
  ```

```
    /*public static*/ class Inner {}
}
final class C /*extends Object*/ {
    /*final*/ void m() {}
}
```

Note that Jikes will generate warnings for redundant modifiers if you use `+Predundant-modifiers` on the command line.

- Modifiers should be listed in the standard order recommended by the JLS. Jikes will warn for this when given `+Pmodifier-order`.
- Because the output of different compilers differs, we have standardized on explicitly specifying `serialVersionUID` in `Serializable` classes in Classpath. This field should be declared as `private static final`. Note that a class may be `Serializable` without being explicitly marked as such, due to inheritance. For instance, all subclasses of `Throwable` need to have `serialVersionUID` declared.
- Don't declare unchecked exceptions in the `throws` clause of a method. However, if throwing an unchecked exception is part of the method's API, you should mention it in the Javadoc. There is one important exception to this rule, which is that a stub method should be marked as throwing `gnu.classpath.NotImplementedException`. This will let our API comparison tools note that the method is not fully implemented.
- When overriding `Object.equals`, remember that `instanceof` filters out `null`, so an explicit check is not needed.
- When catching an exception and rethrowing a new exception you should "chain" the Throwables. Don't just add the String representation of the caught exception.

```
try
  {
    // Some code that can throw
  }
catch (IOException ioe)
  {
    throw (SQLException) new SQLException("Database corrupt").setCause(ioe);
  }
```

- Avoid the use of reserved words for identifiers. This is obvious with those such as `if` and `while` which have always been part of the Java programming language, but you should be careful about accidentally using words which have been added in later versions. Notable examples are `assert` (added in 1.4) and `enum` (added in 1.5). Jikes will warn of the use of the word `enum`, but, as it doesn't yet support the 1.5 version of the language, it will still allow this usage through. A compiler which supports 1.5 (e.g. the Eclipse compiler, ecj) will simply fail to compile the offending source code.

Some things are the same as in the normal GNU Coding Standards:

- Unnecessary braces can be removed, one line after an if, for, while as examples.
- Space around operators (assignment, logical, relational, bitwise, mathematical, shift).
- Blank line before single-line comments, multi-line comments, javadoc comments.
- If more than 2 blank lines, trim to 2.
- Don't keep commented out code. Just remove it or add a real comment describing what it used to do and why it is changed to the current implementation.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

10. Working on the code, Working with others

There are a lot of people helping out with GNU Classpath. Here are a couple of practical guidelines to make working together on the code smoother.

The main thing is to always discuss what you are up to on the mailinglist. Making sure that everybody knows who is working on what is the most important thing to make sure we cooperate most effectively.

We maintain a Task List which contains items that you might want to work on.

Before starting to work on something please make sure you read this complete guide. And discuss it on list to make sure your work does not duplicate or interferes with work someone else is already doing. Always make sure that you submit things that are your own work. And that you have paperwork on file (as stated in the requirements section) with the FSF authorizing the use of your additions.

Technically the GNU Classpath project is hosted on Savannah a central point for development, distribution and maintenance of GNU Software. Here you will find the project page, bug reports, pending patches, links to mailing lists, news items and CVS.

You can find instructions on getting a CVS checkout for classpath at https://savannah.gnu.org/cvs/?group=classpath.

You don't have to get CVS commit write access to contribute, but it is sometimes more convenient to be able to add your changes directly to the project CVS. Please contact the GNU Classpath savannah admins to arrange CVS access if you would like to have it.

Make sure to be subscribed to the commit-classpath mailinglist while you are actively hacking on Classpath. You have to send patches (cvs diff -uN) to this list before committing.

We really want to have a pretty open check-in policy. But this means that you should be extra careful if you check something in. If at all in doubt or if you think that something might need extra explaining since it is not completely obvious please make a little announcement about the change on the mailinglist. And if you do commit something without discussing it first and another GNU Classpath hackers asks for extra explanation or suggests to revert a certain commit then please reply to the request by explaining why something should be so or if you agree to revert it. (Just reverting immediately is OK without discussion, but then please don't mix it with other changes and please say so on list.)

Patches that are already approved for libgcj or also OK for Classpath. (But you still have to send a patch/diff to the list.) All other patches require you to think whether or not they are really OK and non-controversial, or if you would like some feedback first on them before committing. We might get real commit rules in the future, for now use your own judgement, but be a bit conservative.

Always contact the GNU Classpath maintainer before adding anything non-trivial that you didn't write yourself and that does not come from libgcj or from another known GNU Classpath or libgcj hacker. If you have been assigned to commit changes on behalf of another project or a company always make sure they come from people who have signed the papers for the FSF and/or fall under the arrangement your company made with the FSF for contributions. Mention in the ChangeLog who actually wrote the patch.

Commits for completely unrelated changes they should be committed separately (especially when doing a formatting change and a logical change, do them in two separate commits). But do try to do a commit of

as much things/files that are done at the same time which can logically be seen as part of the same change/cleanup etc.

When the change fixes an important bug or adds nice new functionality please write a short entry for inclusion in the `NEWS' file. If it changes the VM interface you must mention that in both the `NEWS' file and the VM Integration Guide.

All the "rules" are really meant to make sure that GNU Classpath will be maintainable in the long run and to give all the projects that are now using GNU Classpath an accurate view of the changes we make to the code and to see what changed when. If you think the requirements are "unworkable" please try it first for a couple of weeks. If you still feel the same after having some more experience with the project please feel free to bring up suggestions for improvements on the list. But don't just ignore the rules! Other hackers depend on them being followed to be the most productive they can be (given the above constraints).

10.1 Working with branches

Sometimes it is necessary to create branch of the source for doing new work that is disruptive to the other hackers, or that needs new language or libraries not yet (easily) available.

After discussing the need for a branch on the main mailinglist with the other hackers explaining the need of a branch and suggestion of the particular branch rules (what will be done on the branch, who will work on it, will there be different commit guidelines then for the mainline trunk and when is the branch estimated to be finished and merged back into the trunk) every GNU Classpath hacker with commit access should feel free to create a branch. There are however a couple of rules that every branch should follow:

- All branches ought to be documented in the developer wiki at http://developer.classpath.org/mediation/ClasspathBranches, so we can know which are live, who owns them, and when they die.
- Some rules can be changed on a branch. In particular the branch maintainer can change the review requirements, and the requirement of keeping things building, testing, etc, can also be lifted. (These should be documented along with the branch name and owner if they differ from the trunk.)
- Requirements for patch email to classpath-patches and for paperwork **cannot** be lifted. See Requirements.
- A branch should not be seen as "private" or "may be completely broken". It should be as much as possible something that you work on with a team (and if there is no team - yet - then there is nothing as bad as having a completely broken build to get others to help out). There can of course be occasional breakage, but it should be planned and explained. And you can certainly have a rule like "please ask me before committing to this branch".
- Merges from the trunk to a branch are at the discretion of the branch maintainer.
- A merge from a branch to the trunk is treated like any other patch. In particular, it has to go through review, it must satisfy all the trunk requirements (build, regression test, documentation).
- There may be additional timing requirements on merging a branch to the trunk depending on the release schedule, etc. For instance we may not want to do a branch merge just before a release.

If any of these rules are unclear please discuss on the list first.

10.2 Documenting what changed when with ChangeLog entries

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

10.2 Documenting what changed when with ChangeLog entries

To keep track of who did what when we keep an explicit ChangeLog entry together with the code. This mirrors the CVS commit messages and in general the ChangeLog entry is the same as the CVS commit message. This provides an easy way for people getting a (snapshot) release or without access to the CVS server to see what happened when. We do not generate the ChangeLog file automatically from the CVS server since that is not reliable.

A good ChangeLog entry guideline can be found in the Guile Manual at http://www.gnu.org/software/guile/changelogs/guile-changelogs_3.html.

Here are some example to explain what should or shouldn't be in a ChangeLog entry (and the corresponding commit message):

- The first line of a ChangeLog entry should be:

  ```
  [date] <two spaces> [full name] <two spaces> [email-contact]
  ```

  The second line should be blank. All other lines should be indented with one tab.

- Just state what was changed. Why something is done as it is done in the current code should be either stated in the code itself or be added to one of the documentation files (like this Hacking Guide).

  So don't write:

  ```
  * java/awt/font/OpenType.java: Remove 'public static final'
  from OpenType tags, reverting the change of 2003-08-11.  See
  Classpath discussion list of 2003-08-11.
  ```

  Just state:

  ```
  * java/awt/font/OpenType.java: Remove 'public static final' from
  all member fields.
  ```

  In this case the reason for the change was added to this guide.

- Just as with the normal code style guide, don't make lines longer then 80 characters.
- Just as with comments in the code. The ChangeLog entry should be a full sentence, starting with a capital and ending with a period.
- Be precise in what changed, not the effect of the change (which should be clear from the code/patch). So don't write:

  ```
  * java/io/ObjectOutputStream.java : Allow putFields be called more
  than once.
  ```

But explain what changed and in which methods it was changed:

```
* java/io/ObjectOutputStream.java (putFields): Don't call
markFieldsWritten(). Only create new PutField when
currentPutField is null.
(writeFields): Call markFieldsWritten().
```

The above are all just guidelines. We all appreciate the fact that writing ChangeLog entries, using a coding style that is not "your own" and the CVS, patch and diff tools do take some time to getting used to. So don't feel like you have to do it perfect right away or that contributions aren't welcome if they aren't "perfect". We all learn by doing and interacting with each other.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

11. Programming Goals

When you write code for Classpath, write with three things in mind, and in the following order: portability, robustness, and efficiency.

If efficiency breaks portability or robustness, then don't do it the efficient way. If robustness breaks portability, then bye-bye robust code. Of course, as a programmer you would probably like to find sneaky ways to get around the issue so that your code can be all three ... the following chapters will give some hints on how to do this.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

11.1 Portability

The portability goal for Classpath is the following:

1. native functions for each platform that work across all VMs on that platform
2. a single classfile set that work across all VMs on all platforms that support the native functions.

For almost all of Classpath, this is a very feasible goal, using a combination of JNI and native interfaces. This is what you should shoot for. For those few places that require knowledge of the Virtual Machine beyond that provided by the Java standards, the VM Interface was designed. Read the Virtual Machine Integration Guide for more information.

Right now the only supported platform is Linux. This will change as that version stabilizes and we begin the effort to port to many other platforms. Jikes RVM runs Classpath on AIX, and generally the Jikes

RVM team fixes Classpath to work on that platform.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

11.2 Utility Classes

At the moment, we are not very good at reuse of the JNI code. There have been some attempts, called *libclasspath*, to create generally useful utility classes. The utility classes are in the directory `native/jni/classpath' and they are mostly declared in `native/jni/classpath/jcl.h'. These utility classes are currently only discussed in Robustness and in Native Efficiency.

There are more utility classes available that could be factored out if a volunteer wants something nice to hack on. The error reporting and exception throwing functions and macros in `native/jni/gtk-peer/gthread-jni.c' might be good candidates for reuse. There are also some generally useful utility functions in `gnu_java_awt_peer_gtk_GtkMainThread.c' that could be split out and put into libclasspath.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

11.3 Robustness

Native code is very easy to make non-robust. (That's one reason Java is so much better!) Here are a few hints to make your native code more robust.

Always check return values for standard functions. It's sometimes easy to forget to check that malloc() return for an error. Don't make that mistake. (In fact, use JCL_malloc() in the jcl library instead-it will check the return value and throw an exception if necessary.)

Always check the return values of JNI functions, or call `ExceptionOccurred` to check whether an error occurred. You must do this after *every* JNI call. JNI does not work well when an exception has been raised, and can have unpredictable behavior.

Throw exceptions using `JCL_ThrowException`. This guarantees that if something is seriously wrong, the exception text will at least get out somewhere (even if it is stderr).

Check for null values of `jclass`es before you send them to JNI functions. JNI does not behave nicely when you pass a null class to it: it terminates Java with a "JNI Panic."

In general, try to use functions in `native/jni/classpath/jcl.h'. They check exceptions and return values and throw appropriate exceptions.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

11.4 Java Efficiency

For methods which explicitly throw a `NullPointerException` when an argument is passed which is null, per a Sun specification, do not write code like:

```
int
strlen (String foo) throws NullPointerException
{
  if (foo == null)
    throw new NullPointerException ("foo is null");
  return foo.length ();
}
```

Instead, the code should be written as:

```
int
strlen (String foo) throws NullPointerException
{
  return foo.length ();
}
```

Explicitly comparing foo to null is unnecessary, as the virtual machine will throw a NullPointerException when length() is invoked. Classpath is designed to be as fast as possible - every optimization, no matter how small, is important.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

11.5 Native Efficiency

You might think that using native methods all over the place would give our implementation of Java speed, speed, blinding speed. You'd be thinking wrong. Would you believe me if I told you that an empty *interpreted* Java method is typically about three and a half times *faster* than the equivalent native method?

Bottom line: JNI is overhead incarnate. In Sun's implementation, even the JNI functions you use once you get into Java are slow.

A final problem is efficiency of native code when it comes to things like method calls, fields, finding classes, etc. Generally you should cache things like that in static C variables if you're going to use them over and over again. GetMethodID(), GetFieldID(), and FindClass() are *slow*. Classpath provides utility libraries for caching methodIDs and fieldIDs in `` `native/jni/classpath/jnilink.h' ``. Other native data can be cached between method calls using functions found in `` `native/jni/classpath/native_state.h' ``.

Here are a few tips on writing native code efficiently:

Make as few native method calls as possible. Note that this is not the same thing as doing less in native method calls; it just means that, if given the choice between calling two native methods and writing a single native method that does the job of both, it will usually be better to write the single native method. You can even call the other two native methods directly from your native code and not incur the overhead of a method call from Java to C.

Cache `jmethodIDs` and `jfieldIDs` wherever you can. String lookups are expensive. The best way to do this is to use the `native/jni/classpath/jnilink.h' library. It will ensure that `jmethodIDs` are always valid, even if the class is unloaded at some point. In 1.1, jnilink simply caches a `NewGlobalRef()` to the method's underlying class; however, when 1.2 comes along, it will use a weak reference to allow the class to be unloaded and then re-resolve the `jmethodID` the next time it is used.

Cache classes that you need to access often. jnilink will help with this as well. The issue here is the same as the methodID and fieldID issue-how to make certain the class reference remains valid.

If you need to associate native C data with your class, use Paul Fisher's native_state library (NSA). It will allow you to get and set state fairly efficiently. Japhar now supports this library, making native state get and set calls as fast as accessing a C variable directly.

If you are using native libraries defined outside of Classpath, then these should be wrapped by a Classpath function instead and defined within a library of their own. This makes porting Classpath's native libraries to new platforms easier in the long run. It would be nice to be able to use Mozilla's NSPR or Apache's APR, as these libraries are already ported to numerous systems and provide all the necessary system functions as well.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

11.6 Security

Security is such a huge topic it probably deserves its own chapter. Most of the current code needs to be audited for security to ensure all of the proper security checks are in place within the Java platform, but also to verify that native code is reasonably secure and avoids common pitfalls, buffer overflows, etc. A good source for information on secure programming is the excellent HOWTO by David Wheeler, Secure Programming for Linux and Unix HOWTO.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

12. API Compatibility

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

12.1 Serialization

Sun has produced documentation concerning much of the information needed to make Classpath serializable compatible with Sun implementations. Part of doing this is to make sure that every class that is Serializable actually defines a field named serialVersionUID with a value that matches the output of serialver on Sun's implementation. The reason for doing this is below.

If a class has a field (of any accessibility) named serialVersionUID of type long, that is what serialver uses. Otherwise it computes a value using some sort of hash function on the names of all method signatures in the .class file. The fact that different compilers create different synthetic method signatures, such as access$0() if an inner class needs access to a private member of an enclosing class, make it impossible for two distinct compilers to reliably generate the same serial #, because their .class files differ. However, once you have a .class file, its serial # is unique, and the computation will give the same result no matter what platform you execute on.

Serialization compatibility can be tested using tools provided with Japitools. These tools can test binary serialization compatibility and also provide information about unknown serialized formats by writing these in XML instead. Japitools is also the primary means of checking API compatibility for GNU Classpath with Sun's Java Platform.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

12.2 Deprecated Methods

Sun has a practice of creating "alias" methods, where a public or protected method is deprecated in favor of a new one that has the same function but a different name. Sun's reasons for doing this vary; as an example, the original name may contain a spelling error or it may not follow Java naming conventions.

Unfortunately, this practice complicates class library code that calls these aliased methods. Library code must still call the deprecated method so that old client code that overrides it continues to work. But library code must also call the new version, because new code is expected to override the new method.

The correct way to handle this (and the way Sun does it) may seem counterintuitive because it means that new code is less efficient than old code: the new method must call the deprecated method, and throughout the library code calls to the old method must be replaced with calls to the new one.

Take the example of a newly-written container laying out a component and wanting to know its preferred size. The Component class has a deprecated preferredSize method and a new method, getPreferredSize. Assume that the container is laying out an old component that overrides preferredSize and a new component that overrides getPreferredSize. If the container calls getPreferredSize and the default implementation of getPreferredSize calls preferredSize, then the old component will have its preferredSize method called and new code will have its getPreferredSize method called.

Even using this calling scheme, an old component may still be laid out improperly if it implements a method, getPreferredSize, that has the same signature as the new Component.getPreferredSize. But that is a general problem - adding new public or protected methods to a widely-used class that calls those methods internally is risky, because existing client code may have already declared methods with the same signature.

The solution may still seem counterintuitive - why not have the deprecated method call the new method, then have the library always call the old method? One problem with that, using the preferred size example again, is that new containers, which will use the non-deprecated getPreferredSize, will not get the preferred size of old components.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

13. Specification Sources

There are a number of specification sources to use when working on Classpath. In general, the only place you'll find your classes specified is in the JavaDoc documentation or possibly in the corresponding white paper. In the case of java.lang, java.io and java.util, you should look at the Java Language Specification.

Here, however, is a list of specs, in order of canonicality:

1. Clarifications and Amendments to the JLS - 1.1
2. JLS Updates - 1.1
3. The 1.0 JLS
4. JVM spec - 1.1
5. JNI spec - 1.1
6. Sun's javadoc - 1.1 (since Sun's is the reference implementation, the javadoc is documentation for the Java platform itself.)
7. JVMDI spec - 1.2, JNI spec - 1.2 (sometimes gives clues about unspecified things in 1.1; if it was not specified accurately in 1.1, then use the spec for 1.2; also, we are using JVMDI in this project.)
8. Sun's javadoc - 1.2 (sometimes gives clues about unspecified things in 1.1; if it was not specified accurately in 1.1, then use the spec for 1.2)
9. The Bug Parade: I have obtained a ton of useful information about how things do work and how they *should* work from the Bug Parade just by searching for related bugs. The submitters are very careful about their use of the spec. And if something is unspecified, usually you can find a request for specification or a response indicating how Sun thinks it should be specified here.

You'll notice that in this document, white papers and specification papers are more canonical than the JavaDoc documentation. This is true in general.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

14. Directory and File Naming Conventions

The Classpath directory structure is laid out in the following manner:

```
classpath
  |
  |---->java
  |        |
  |        |-->awt
  |        |-->io
  |        |-->lang
  |        |-->util
  |        |     |
  |        |     |--->zip
  |        |     |--->jar
  |        |-->net
  |        |-->etc
  |
  |---->gnu
  |        |
  |        |-->java
```

```
|              |
|              |-->awt
|              |-->lang
|              |-->util
|              |      |
|              |      |-->zip
|              |-->etc
|
|----->native
         |
         |-->jni
         |     |-->classpath
         |     |-->gtk-peer
         |     |-->java-io
         |     |-->java-lang
         |     |-->java-net
         |     |-->java-util
         |     |-->etc
         |-->cni
```

Here is a brief description of the toplevel directories and their contents.

java

Contains the source code to the Java packages that make up the core class library. Because this is the public interface to Java, it is important that the public classes, interfaces, methods, and variables are exactly the same as specified in Sun's documentation. The directory structure is laid out just like the java package names. For example, the class java.util.zip would be in the directory java-util.

gnu/java

Internal classes (roughly analogous to Sun's sun.* classes) should go under the `gnu/java' directory. Classes related to a particular public Java package should go in a directory named like that package. For example, classes related to java.util.zip should go under a directory `gnu/java/util/zip'. Sub-packages under the main package name are allowed. For classes spanning multiple public Java packages, pick an appropriate name and see what everybody else thinks.

native

This directory holds native code needed by the public Java packages. Each package has its own subdirectory, which is the "flattened" name of the package. For example, native method implementations for java.util.zip should go in `native/classpath/java-util'. Classpath actually includes an all Java version of the zip classes, so no native code is required.

Each person working on a package get's his or her own "directory space" underneath each of the toplevel directories. In addition to the general guidelines above, the following standards should be followed:

- Classes that need to load native code should load a library with the same name as the flattened package name, with all hyphens removed. For example, the native library name specified in LoadLibrary for java-util would be "javautil".

- Each package has its own shared library for native code (if any).
- The main native method implementation for a given method in class should go in a file with the same name as the class with a ".c" extension. For example, the JNI implementation of the native methods in java.net.InetAddress would go in `native/jni/java-net/InetAddress.c'. "Internal" native functions called from the main native method can reside in files of any name.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

15. Character Conversions

Java uses the Unicode character encoding system internally. This is a sixteen bit (two byte) collection of characters encompassing most of the world's written languages. However, Java programs must often deal with outside interfaces that are byte (eight bit) oriented. For example, a Unix file, a stream of data from a network socket, etc. Beginning with Java 1.1, the `Reader` and `Writer` classes provide functionality for dealing with character oriented streams. The classes `InputStreamReader` and `OutputStreamWriter` bridge the gap between byte streams and character streams by converting bytes to Unicode characters and vice versa.

In Classpath, `InputStreamReader` and `OutputStreamWriter` rely on an internal class called `gnu.java.io.EncodingManager` to load translators that perform the actual conversion. There are two types of converters, encoders and decoders. Encoders are subclasses of `gnu.java.io.encoder.Encoder`. This type of converter takes a Java (Unicode) character stream or buffer and converts it to bytes using a specified encoding scheme. Decoders are a subclass of `gnu.java.io.decoder.Decoder`. This type of converter takes a byte stream or buffer and converts it to Unicode characters. The `Encoder` and `Decoder` classes are subclasses of `Writer` and `Reader` respectively, and so can be used in contexts that require character streams, but the Classpath implementation currently does not make use of them in this fashion.

The `EncodingManager` class searches for requested encoders and decoders by name. Since encoders and decoders are separate in Classpath, it is possible to have a decoder without an encoder for a particular encoding scheme, or vice versa. `EncodingManager` searches the package path specified by the `file.encoding.pkg` property. The name of the encoder or decoder is appended to the search path to produce the required class name. Note that `EncodingManager` knows about the default system encoding scheme, which it retrieves from the system property `file.encoding`, and it will return the proper translator for the default encoding if no scheme is specified. Also, the Classpath standard translator library, which is the `gnu.java.io` package, is automatically appended to the end of the path.

For efficiency, `EncodingManager` maintains a cache of translators that it has loaded. This eliminates the need to search for a commonly used translator each time it is requested.

Finally, `EncodingManager` supports aliasing of encoding scheme names. For example, the ISO Latin-1 encoding scheme can be referred to as "8859_1" or "ISO-8859-1". `EncodingManager` searches for aliases by looking for the existence of a system property called `gnu.java.io.encoding_scheme_alias.<encoding name>`. If such a property exists. The value of that property is assumed to be the canonical name of the encoding scheme, and a translator with that name is looked up instead of one with the original name.

Here is an example of how `EncodingManager` works. A class requests a decoder for the "UTF-8" encoding scheme by calling `EncodingManager.getDecoder("UTF-8")`. First, an alias is searched for by

looking for the system property `gnu.java.io.encoding_scheme_alias.UTF-8`. In our example, this property exists and has the value "UTF8". That is the actual decoder that will be searched for. Next, `EncodingManager` looks in its cache for this translator. Assuming it does not find it, it searches the translator path, which is this example consists only of the default `gnu.java.io`. The "decoder" package name is appended since we are looking for a decoder. ("encoder" would be used if we were looking for an encoder). Then name name of the translator is appended. So `EncodingManager` attempts to load a translator class called `gnu.java.io.decoder.UTF8`. If that class is found, an instance of it is returned. If it is not found, a `UnsupportedEncodingException`.

To write a new translator, it is only necessary to subclass `Encoder` and/or `Decoder`. Only a handful of abstract methods need to be implemented. In general, no methods need to be overridden. The needed methods calculate the number of bytes/chars that the translation will generate, convert buffers to/from bytes, and read/write a requested number of characters to/from a stream.

Many common encoding schemes use only eight bits to encode characters. Writing a translator for these encodings is very easy. There are abstract translator classes `gnu.java.io.decode.DecoderEightBitLookup` and `gnu.java.io.encode.EncoderEightBitLookup`. These classes implement all of the necessary methods. All that is necessary to create a lookup table array that maps bytes to Unicode characters and set the class variable `lookup_table` equal to it in a static initializer. Also, a single constructor that takes an appropriate stream as an argument must be supplied. These translators are exceptionally easy to create and there are several of them supplied in the Classpath distribution.

Writing multi-byte or variable-byte encodings is more difficult, but often not especially challenging. The Classpath distribution ships with translators for the UTF8 encoding scheme which uses from one to three bytes to encode Unicode characters. This can serve as an example of how to write such a translator.

Many more translators are needed. All major character encodings should eventually be supported.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

16. Localization

There are many parts of the Java standard runtime library that must be customized to the particular locale the program is being run in. These include the parsing and display of dates, times, and numbers; sorting words alphabetically; breaking sentences into words, etc. In general, Classpath uses general classes for performing these tasks, and customizes their behavior with configuration data specific to a given locale.

In Classpath, all locale specific data is stored in a `ListResourceBundle` class in the package `gnu/java/locale`. The basename of the bundle is `LocaleInformation`. See the documentation for the `java.util.ResourceBundle` class for details on how the specific locale classes should be named.

`ListResourceBundle`'s are used instead of `PropertyResourceBundle`'s because data more complex than

simple strings need to be provided to configure certain Classpath components. Because `ListResourceBundle` allows an arbitrary Java object to be associated with a given configuration option, it provides the needed flexibility to accomodate Classpath's needs.

Each Java library component that can be localized requires that certain configuration options be specified in the resource bundle for it. It is important that each and every option be supplied for a specific component or a critical runtime error will most likely result.

As a standard, each option should be assigned a name that is a string. If the value is stored in a class or instance variable, then the option should name should have the name name as the variable. Also, the value associated with each option should be a Java object with the same name as the option name (unless a simple scalar value is used). Here is an example:

A class loads a value for the `format_string` variable from the resource bundle in the specified locale. Here is the code in the library class:

```
ListResourceBundle lrb =
  ListResourceBundle.getBundle ("gnu/java/locale/LocaleInformation", locale);
String format_string = lrb.getString ("format_string");
```

In the actual resource bundle class, here is how the configuration option gets defined:

```
/**
  * This is the format string used for displaying values
  */
private static final String format_string = "%s %d %i";

private static final Object[][] contents =
{
  { "format_string", format_string }
};
```

Note that each variable should be `private`, `final`, and `static`. Each variable should also have a description of what it does as a documentation comment. The `getContents()` method returns the `contents` array.

There are many functional areas of the standard class library that are configured using this mechanism. A given locale does not need to support each functional area. But if a functional area is supported, then all of the specified entries for that area must be supplied. In order to determine which functional areas are supported, there is a special key that is queried by the affected class or classes. If this key exists, and has a value that is a `Boolean` object wrappering the `true` value, then full support is assumed. Otherwise it is assumed that no support exists for this functional area. Every class using resources for configuration must use this scheme and define a special scheme that indicates the functional area is supported. Simply checking for the resource bundle's existence is not sufficient to ensure that a given functional area is supported.

The following sections define the functional areas that use resources for locale specific configuration in GNU Classpath. Please refer to the documentation for the classes mentioned for details on how these values are used. You may also wish to look at the source file for `` `gnu/java/locale/LocaleInformation_en' `` as an example.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

16.1 String Collation

Collation involves the sorting of strings. The Java class library provides a public class called `java.text.RuleBasedCollator` that performs sorting based on a set of sorting rules.

- RuleBasedCollator - A `Boolean` wrappering `true` to indicate that this functional area is supported.
- collation_rules - The rules the specify how string collation is to be performed.

Note that some languages might be too complex for `RuleBasedCollator` to handle. In this case an entirely new class might need to be written in lieu of defining this rule string.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

16.2 Break Iteration

The class `java.text.BreakIterator` breaks text into words, sentences, and lines. It is configured with the following resource bundle entries:

- BreakIterator - A `Boolean` wrappering `true` to indicate that this functional area is supported.
- word_breaks - A `String` array of word break character sequences.
- sentence_breaks - A `String` array of sentence break character sequences.
- line_breaks - A `String` array of line break character sequences.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

16.3 Date Formatting and Parsing

Date formatting and parsing is handled by the `java.text.SimpleDateFormat` class in most locales. This class is configured by attaching an instance of the `java.text.DateFormatSymbols` class. That class simply reads properties from our locale specific resource bundle. The following items are required (refer to the documentation of the `java.text.DateFormatSymbols` class for details io what the actual values should be):

- DateFormatSymbols - A `Boolean` wrappering `true` to indicate that this functional area is supported.
- months - A `String` array of month names.
- shortMonths - A `String` array of abbreviated month names.
- weekdays - A `String` array of weekday names.
- shortWeekdays - A `String` array of abbreviated weekday names.
- ampms - A `String` array containing AM/PM names.
- eras - A `String` array containing era (i.e., BC/AD) names.
- zoneStrings - An array of information about valid timezones for this locale.

- localPatternChars - A `String` defining date/time pattern symbols.
- shortDateFormat - The format string for dates used by `DateFormat.SHORT`
- mediumDateFormat - The format string for dates used by `DateFormat.MEDIUM`
- longDateFormat - The format string for dates used by `DateFormat.LONG`
- fullDateFormat - The format string for dates used by `DateFormat.FULL`
- shortTimeFormat - The format string for times used by `DateFormat.SHORT`
- mediumTimeFormat - The format string for times used by `DateFormat.MEDIUM`
- longTimeFormat - The format string for times used by `DateFormat.LONG`
- fullTimeFormat - The format string for times used by `DateFormat.FULL`

Note that it may not be possible to use this mechanism for all locales. In those cases a special purpose class may need to be written to handle date/time processing.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

16.4 Decimal/Currency Formatting and Parsing

`NumberFormat` is an abstract class for formatting and parsing numbers. The class `DecimalFormat` provides a concrete subclass that handles this is in a locale independent manner. As with `SimpleDateFormat`, this class gets information on how to format numbers from a class that wrappers a collection of locale specific formatting values. In this case, the class is `DecimalFormatSymbols`. That class reads its default values for a locale from the resource bundle. The required entries are:

- DecimalFormatSymbols - A `Boolean` wrappering `true` to indicate that this functional area is supported.
- currencySymbol - The string representing the local currency.
- intlCurrencySymbol - The string representing the local currency in an international context.
- decimalSeparator - The character to use as the decimal point as a `String`.
- digit - The character used to represent digits in a format string, as a `String`.
- exponential - The char used to represent the exponent separator of a number written in scientific notation, as a `String`.
- groupingSeparator - The character used to separate groups of numbers in a large number, such as the "," separator for thousands in the US, as a `String`.
- infinity - The string representing infinity.
- NaN - The string representing the Java not a number value.
- minusSign - The character representing the negative sign, as a `String`.
- monetarySeparator - The decimal point used in currency values, as a `String`.
- patternSeparator - The character used to separate positive and negative format patterns, as a `String`.
- percent - The percent sign, as a `String`.
- perMill - The per mille sign, as a `String`.
- zeroDigit - The character representing the digit zero, as a `String`.

Note that several of these values are an individual character. These should be wrappered in a `String` at character position 0, not in a `Character` object.

[Top] [Contents] [Index] [?]

About This Document

This document was generated by *Andrew John Hughes* on *February, 6 2009* using *texi2html 1.76*.

The buttons in the navigation panels have the following meaning:

Button	Name	Go to	From 1.2.3 go to
[<]	Back	previous section in reading order	1.2.2
[>]	Forward	next section in reading order	1.2.4
[<<]	FastBack	beginning of this chapter or previous chapter	1
[Up]	Up	up section	1.2
[>>]	FastForward	next chapter	2
[Top]	Top	cover (top) of document	
[Contents]	Contents	table of contents	
[Index]	Index	index	
[?]	About	about (help)	

where the **Example** assumes that the current position is at **Subsubsection One-Two-Three** of a document of the following structure:

- 1. Section One
  - 1.1 Subsection One-One
    - ...
  - 1.2 Subsection One-Two
    - 1.2.1 Subsubsection One-Two-One
    - 1.2.2 Subsubsection One-Two-Two
    - 1.2.3 Subsubsection One-Two-Three <== **Current Position**
    - 1.2.4 Subsubsection One-Two-Four
  - 1.3 Subsection One-Three
    - ...
  - 1.4 Subsection One-Four

This document was generated by *Andrew John Hughes* on *February, 6 2009* using *texi2html 1.76*.